Object-Oriented Design of Graph Oriented Data Structures[*]

(Extended Abstract)

Maurizio Pizzonia[‡] *Giuseppe Di Battista*[‡]

pizzonia@dia.uniroma3.it gdb@dia.uniroma3.it

Abstract

Applied research in graph algorithms and combinatorial structures needs comprehensive and versatile software libraries. However, the design and the implementation of flexible libraries are challenging activities. Among the other problems involved in such a difficult field, a very special role is played by graph classification issues.

We propose new techniques devised to help the designer and the programmer in the development activities. Such techniques are especially suited for dealing with graph classification problems and rely on an extension of the usual object-oriented paradigm. In order to support the usage of our approach, we devised an extension of the C++ programming language and implemented the corresponding pre-compiler.

1 Introduction

Libraries dealing with combinatorial structures like graphs are quite often large systems because of the great variety of known algorithms and because of the possibility of graphs to be classified in a plethora of ways. Also, most of the existing algorithms require a large amount of software to be implemented. To give an example, the GDToolkit [20] system consists of more than $40,000$ lines of code. A limited list of object-oriented systems somehow related to graphs includes: ALF [5], ffgraph [6], JDSL [15], Leda [10], and LINK [4].

Modern large software systems are usually built using object-oriented technologies and methodologies. The main motivations of this choice are: development and maintenance time reduction, reusability of components, and extensibility of the system. A key issue here is to increase the abstraction level of the code by modeling the reality of interest with high-level representation structures.

The above mentioned goals are especially difficult to meet in the application domain that we are considering. Our opinion is that one of the main reasons for this difficulty is in the lack, in currently used object oriented methods and languages, of representation primitives suitable for representing complex combinatorial structures. The experience of designing a large library of graph algorithms offers a clear perception of this type of modeling problems, that are especially related to the classification of graphs. We give an overview and an analysis of such problems in Section 2.

The main contributions of this paper can be summarized as follows:

- We propose an extension of the object-oriented paradigm, specifically tailored to tackle problems arising in the classification of graphs (Section 3). Such an extension (called *ECO*) consists of new representation primitives.
- We show how to use ECO to build large graph libraries. This is done by suggesting several design schemas. A project experience conducted with such design schemas has put in evidence an improved flexibility of the system and a decrease of the development time. The saving of time was mainly due to the high abstraction level of the produced code (Section 4).
- We describe an extension of the C++ programming language that embodies the ECO primitives. Further, we present a pre-compiler that we have implemented for ECO, which gives an easy way to use the concepts of the new paradigm (Section 5).

[*]Research supported in part by the ESPRIT LTR Project no. 20244 - ALCOM-IT.

[‡]Dipartimento di Informatica e Automazione, Università di Roma Tre, via della Vasca Navale 79, 00146 Roma, Italy.

2 Classification Problems for Graphs

Classification is a key part in software libraries that manage combinatorial structures like graphs. In fact, each graph algorithm is suited to work on a specified class of graphs for which given properties hold. However, notwithstanding its importance, the actual role of graph classification is usually reduced by the flexibility and the extensibility problems that a large hierarchy of graph classes introduces (see, e.g. [5]).

In the following we briefly analyze the most, in our opinion, relevant aspects of the interplay between graph classification problems and the object-oriented paradigm.

2.1 Inheritance and Subclasses

Various authors point out that saying that B is a subclass[1] of A can have several meanings (for example see [12, 17]).

We refer to *extension* subclassing if new information is added to B and each instance of A can become an instance of B if suitably equipped with new information (e.g. in this sense Labeled Graph is a subclass of Graph). In the following we call *extension* the "added part" of B and *support* the "inherited part". This is very different from the case where no new information is added to B and there are instances of A that are not instances of B, i.e. B is a strict subclass of A in the mathematical sense (e.g. Connected Graph is a subclass of Graph). In this case we talk about *restriction* subclassing.

The inheritance mechanisms of most object-oriented languages are well suited in modeling extension subclassing, but fail in modeling restriction subclassing. Namely, the restriction subclassing infringes the soundness of polymorphism because not all values that are legal for objects that belong to A are also legal for objects that belong to B. I.e. B violates the *substitution principle* (first stated in [9]). Hence, the programmer must either exploit the exception mechanism when the invariant of B is violated, or, even worse, must rely on the user to behave correctly (such policies are used in libraries like Leda [10], JDSL [15], LINK [4], and GDToolkit [20]). Such sort of situations induce the programmer to modify A in order to consider particular situations that are actually related to B, thus violating another well known principle, the *open-closed principle*: "every module must be closed to modification and open to extensions" (first stated in [11]).

Consequently, restriction subclassing cannot be modelled in any elegant way using standard inheritance mechanisms, unless we make use of the functional-style design[2], which is known to be rather inefficient.

2.2 Crossed Classifications

Complex combinatorial structures like graphs can be classified in a large number of ways. If more than one independent classification exist, *crossed* classes must be created to give to the system complete classification capabilities. For example, Fig. 1 shows how combining connectivity, planarity, and orientation may give raise to several crossed classes, even if none of such classes adds new features to the inherited ones. Of course, the number of crossed classes increases exponentially with the classification coordinates.

This problem is especially relevant because usual object-oriented paradigms do not allow objects to belong to more than one class. The only exception is the usage of multiple inheritance that, however, is not helpful in decreasing the number of subclasses.

Several authors address this problem. For example, [12] uses a "delegation technique", while [7] uses the "decorator pattern". However, the proposed solutions always show an overwhelming burden for the programmer and/or a quite tricky game of conventions and limitations that cannot be expressed in a programming language.

2.3 Multiple Decorations

There are two types of subclasses defined with the extension subclassing.

1. The added information is (if needed) univocally determined by the support. For example, if the superclass is a graph, then the subclass might be a graph equipped with the set of the connected components. Given a graph, such information is univocally determined.

[1]We do not distinguish between classes and types (see [1]) because it is not relevant to our discussion. However, what we say can be easily extended to handle such distinction.

[2]A functional-style design avoids any changes to existing objects; methods that change the graph actually produce a new copy of the object. The class of the new copy can also be different from the starting class.

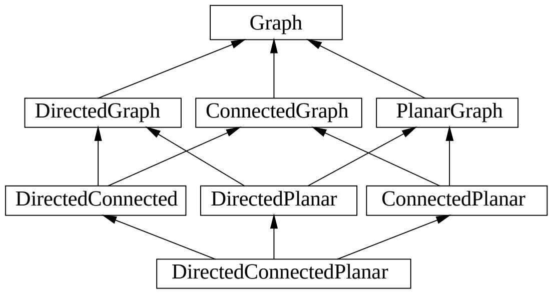

Figure 1: A hierarchy affected by the crossed classification problem.

2. The added information is not univocally determined by the support. For example, an st-numbering of a graph is (at least partially) arbitrary.

In Case 1, it is reasonable to exploit the inheritance for adding the information about the set of the connected components.

Case 2 is simple if we do not need to add more information (e.g. other st-numberings or orientations). If this is true, then the usage of the inheritance remains a reasonable choice. However, if we require the support to be extended with other information of the same type, then it is not possible to use the inheritance. In other words, referring to the example, the inheritance forces to specify just one st-numbering, while some applications may require to st-number the vertices of the graph in several ways at the same time.

A possible solution is to instantiate one new object for each new information (e.g. for each st-numbering) which contains the new information. It is also needed to state links between elements of the first object (e.g. the vertices) and the new objects (e.g. the numerical labels) by means of suitable structures (e.g. hash-tables). Even though many libraries provide mechanisms to state such links, it may become difficult (or even practically impossible) to maintain the consistency among the objects when the graph changes.

2.4 Promotion Efficiency

During the execution of an algorithm a graph may change its properties. This has the effect of virtually moving the graph to another class, where new methods become meaningful. Further, even if the properties of a graph do not change, it is sometimes useful, for efficency reasons, to dynamically equip the graph with new capabilities when they are needed.

In general, when an object is *promoted* (*demoted*) from a class into its child (parent), it requires a complete copy of the support. For example, if the graph is recognized to be planar, and we decide to promote it into the suitable class to take advantage of the new properties, a complete copy of the graph (with its “heavy” implementation structures) must be done. If the internal implementation is the same (e.g. linked), a large amount of time is wasted to make the copy. This happens because the membership relation is usually not dynamic in the available programming languages. The class of an object is statically defined at the moment of its creation.

This problem can be addressed by means of the “bridge pattern” [7]. This technique keeps separated the interface and the implementation structures using two distinct objects linked by a pointer. In this way, it is very efficient, although not very elegant, “to steal” the implementation from a graph to create another graph that has an extended interface. This technique is used in GDToolkit [20].

3 New Representation Primitives for Graphs Classification

The *ECO* (Extender and Classer Oriented) paradigm is an extension of the usual object-oriented design paradigm. This means that ECO imports the set of representation primitives usually defined in object-oriented design methodologies and enlarges such a set with new concepts. The new concepts are expecially targeted to address the problems illustrated in Section 2.

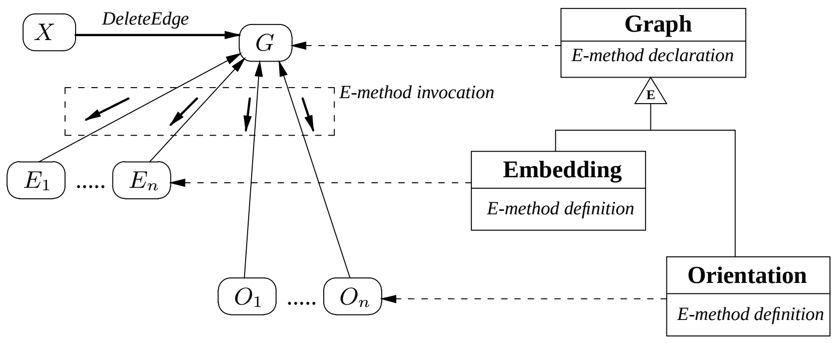

Figure 2: In this mixed (classes-objects) diagram, dashed lines represent instance relationships (E_i are embeddings and O_j are orientations), thick arrows represent method (or E-method) invocations, plain lines represent extension relationships. When a method is invoked on a Graph G (for example *DeleteEdge*), the E-method mechanism can be used to notify the event to the current extender-objects.

3.1 Extenders and E-Methods

The *extender* is the main new concept introduced by ECO. Extenders are classes that have the following additional features. An extender is associated to a *support-class*. An instance of an extender, called *extension-object*, is created over a *support-object* belonging to the support-class. It must be created after the support-object and it must be destroyed before the support-object. Thus, we can talk about the current extension-objects of a given support-object. Given a support-class C and an extender E for C, it is possible for an instance of C to be support for more than one instance of E. To give an example, a class Graph can be support-class for the extenders Orientation and Embedding, so, more than one orientation and/or embedding may be instantiated over a given graph.

When an event occurs, then the support-object can notify it to its current extension-objects. The notification is done by using specific methods, called *E-methods*. An E-method signature is defined in a support-class and the behavior is specified in its extenders. An E-method equips an extension-object with two capabilities. An extension-object

- can change its state when the state of the support-object changes,
- can add constraints to the possible changes of the state of the support-object (using exception mechanisms).

E-methods are invoked only in the methods of the support-class. The invocation of an E-method triggers the execution of its behavior for each current extension-object (see Fig. 2). The behavior of the E-methods is allowed to modify only the state of the extension-object, which it is invoked for. The executions sequence is not specified. Intuitively, they can be considered parallel executions, while the support-object waits for the termination of all of them.

Fig. 3 shows the evolution of a system in which a graph G (a support-object) and some embedding and orientation $E_1, \ldots, E_n, O_1, \ldots, O_m$ for G (extension-objects) interact. The vertical direction represents the time. Each vertical line represents the evolution of an object. A vertical line becomes thick when a method is executed on the corresponding object. The E-method Post_AddVertex(v) is invoked from a method AddVertex( ) of G. This triggers the execution on each embedding and each orientation of the behavior of Post_AddVertex( ) associated to such objects. Observe that distinct extension-objects may have a distinct behavior depending on the extenders they belong to. So, in the example, orientations can have a behavior that is different from the behavior of embeddings.

The E-methods are most likely to be used in methods that change the state of the support-object. In fact, such changes can lead to inconsistencies in the current extension-objects. E-methods can update the extension-object or inhibith the change in the support, depending on design choices.

For each method that performs modifications on the state of the support-object the designer can provide an E-method that is called as soon as the method is called. The definitions of these E-methods are supposed to inhibit the modification (i.e. throwing an exception) if the extender constraints are violated. Two more E-methods can be provided in order to allow each extension-object to update its state on legal modifications of the graph. These are called before and after the modification of the support-object.

Extenders are especially suited to support the extension subclassing where the multiple decoration problem arises (see Section 2.3). They are useful to dynamically change behavioral and/or structural aspects of an object during its

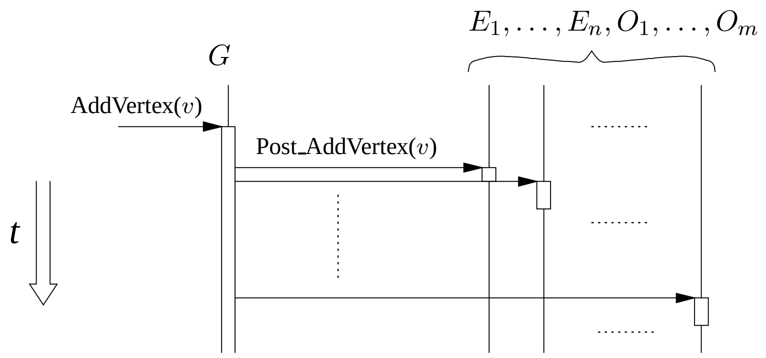

Figure 3: Interaction between a support-object G and its extension-objects $E_1, \ldots, E_n, O_1, \ldots, O_m$.

life-time. Further, an extender can be developed even much time after the development of its support-class. This allows to add capabilities to the system without changing any other part of it. The new extender will coexist with other extenders already present in the system, and will work in conjunction with them.

3.2 Classers

The *classer* is the second new concept. A classer is a constrained extender for which just one instance is allowed for a given support-object[3]. In spite of the simplicity of the definition, classers are as important as extenders and give much more expressiveness to the paradigm.

Consider the case when new information, and operations related to it, have to be added to an object and such information can be univocally determined by its state by means of a functional dependency (for example the set of connected components, see Section 2.3). Also, consider the need of attaching the new information dynamically. In this case a dynamic classification system would be needed allowing an object to change its class during its life-time. Even if extenders seem to be a good choice to support dynamic classification issues, their capability of having more than one instance, for each support-object, yields several disadvantages.

The main problem is that, under the above conditions, all the extension-objects of a given support-object have the same state, because the information that they contain functionally depends on the state of the support-object. Single extension-objects cannot change independently because of their peculiar semantic. So, having more than one instance is not useful, inefficient, and conceptually misleading.

Then, we can state that the new information and operations are conceptually bound to the support-object. Nevertheless, the programmer has to refer explicitly to the extension-objects using variables or pointers, as he/she refers to any other object. Also, when classification hierarchies are non-trivial the programmer has to deal with much more objects than he/she needs.

Classers are extenders that can have only one instance for each support-object. Hence, the underlying extension-object can be accessed through the support-object itself without maintaining an explicit reference to it. Also, using classers avoids the risk of inefficiency due to multiple instantiations.

Classers are especially suited to support the restriction subclassing and the extension subclassing in the case of information univocally determined by the state of the support-object (see Section 2.3). We would like to point out that using classers in such situations avoids the crossed classification problem and the promotion efficiency problem.

4 Using the ECO Paradigm

In this section we present some techniques especially suited to build large systems using the ECO paradigm. We provide some examples that show how ECO addresses the problems mentioned in Section 2 while respecting the open-closed principle and keeping extendibility and flexibility.

[3]It is important not to misidentify the constraint for the classer with the *singleton* pattern [7]. More than one instance is admitted for classers overall, but only one for each support-object.

4.1 Using Classers to Add Structure

A library for combinatorial mathematics cannot handle all the needs that the user could ask. So, it has to provide powerful and flexible ways to extend its capabilities without modifying the library itself.

Many features that a non naive user can ask require addictional structures to be maintained with a graph and updated when the graph changes (for example, connected components set, block cut-vertex tree, etc.). Such kind of structures are not basic features (there are many applications that do not require them) and could be an unnecessary burden if they are not really needed. For this reason, we prefer not to insert them into the main graph class.

Classers are particularly useful to address such problem. In fact, using classers yields the following advantages:

- the new information can be attached when needed, so, the user is not compelled to choose a cumbersome implementation at the instantiation time;
- the E-methods mechanism provides a clean way to dynamically update the structure and permits to place the updating code in the classer definition;
- the introduction of new classers does not require changes to old code and the new and the old classers can be used simultaneously without any sort of limitation, thus, obtaining good extendibility and flexibility.

Following the presented approach, classers turn out to be the natural containers for dynamic algorithms (for a similar, but ad hoc, approach see [2]).

4.2 Hierarchies of Classers and Extenders

In Section 3.1, we already provided an example of using extenders to model the concepts of orientation and embedding (permitting more instances at once of both), and in Section 4.1 we described how to use classers. Now, we describe how each extender and classer can be a support-class, as well, accepting its own extenders and classers, and thus, allowing even more expressiveness.

An extender can be used to associate new structures to an embedding in order, for example, to represent the orthogonal shape of the edges[4], and an orientations can be equipped with costs and capacities to became a flow network in conjunction with its support graph.

Using this technique the extender-objects can make up a tree. When a support-object changes its state, the consistancy in the lower levels of the extension-objects tree have to be maintained. E-methods can be easily used to address this problem.

The depth of such tree is usually not greater then three or four levels, and constant in any case (it is statically determined at compile time). On the other hand, there is no constraint to the degree of its nodes (i.e. the number of extension-objects). However, note that the number of extension-objects for a given support-object are very often constant for a given algorithm and, more precisely, it does not depend on the input size[5]. Due to the above considerations, the paradigm allows the update to be performed in constant time for a given algorithm.

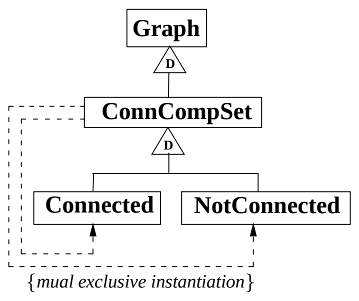

Figure 4: ConnCompSet is a classer that dynamically manage the set of connected components. Connected and NotConnected classers represent the status of the graph. They act like an "hook" for attaching new classers or extenders that deals only with graphs for which the relative property holds.

The classers can also be used to represent restriction subclassing. In fact, we can associate a classer to a property and the presence of a classer instance means that the given property holds for the graph, hence, the graph belong to a specific mathematical class. In Fig. 4 we shows a design schema that permits to dynamically attach the connected component set structure, and then (only if such feature is present) to classify the graph as Connected or NotConnected. The instantiation of such classers can be ascribed to the ConnCompSet classer itself. In this way, we can

[4]In the Graph Drawing area, the sequence of left/right bends of an orthogonal drawing. The shape of a graph plays an important role in algorithms like GIOTTO [14].

[5]Using a set of extension-objects whose size grows with the input size is almost always a misuse. In fact, using extenders implies dynamical update through E-methods, which, in this case, takes linear time. Note that if we really need to update such a set of objects dynamically, extenders largely simplify the work and do not increase the computational complexity.

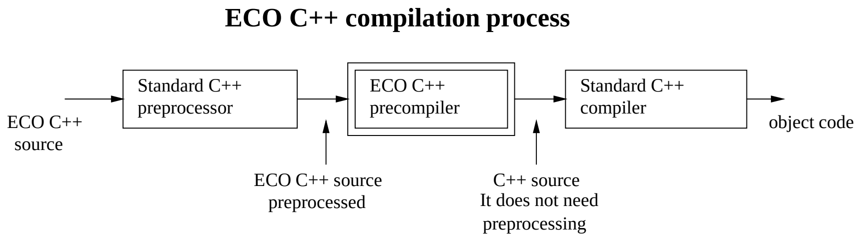
ECO C++ compilation process
ECO C++ source
Standard C++ preprocessor
ECO C++ precompiler
Standard C++ compiler
object code
ECO C++ source preprocessed
C++ source It does not need preprocessing

Figure 5: The ECO C++ compilation process. A pre-compilation phase has been inserted after the preprocessing phase and before the usual compilation phase.

provide a full-fledged module to handle connectivity with automatic dynamic classification[6], that also permits further restriction subclassing (attaching classers to Connected or NotConnected), or extenders that can futher exploit the concept of connected component (attaching extenders to ConnCompSet).

5 Supporting the ECO Paradigm with a Pre-Compiler

The introduction of a new paradigm is of little help if not supported by suitable programming tools. Among several possible alternatives, we have chosen to support the ECO paradigm with a pre-compiler. Also, we introduced an extension of the C++ language which we call ECO C++. The pre-compiler generates code where new constructs are replaced by standard C++ code that emulates their semantic. Fig. 5 shows the entire compilation process.

This yields a number of advantages. The C++ language is a well know language with many libraries and tools already available and widely used. A pre-compiler permits to use the new paradigm without limiting the usage of such existing tools. Further, it does not require the programmer to learn a completely new language, and because of the existence of good, portable and freely available compilers, the system works on a wide range of platforms.

5.1 The ECO C++ language

ECO C++ supports the concept of extenders and E-methods by means of new syntactic primitives. The programmer can declare a support-class and an extender by using a suitable syntax as in the example shown in Fig. 6. The keyword **extensible** denotes a declaration of a support-class in which some E-methods can be declared. E-methods are denoted by the keyword **extend**. The behavior for such E-methods may be defined for each extender (the default behavior is "do nothing"). The E-methods invocation is performed by using the keyword **call_e_method( )** as shown in the same figure. When the control flow reaches such keyword the execution is dispatched to the behaviors specified in each current extension-object.

The keyword **extend** denotes an extender for a specified class, when placed in the head of a class declaration. Each constructor for the extender must have, as first parameter, a reference or a pointer to the support-class. The actual parameter will be the support-object of the extender-object that is being creating (see Fig. 6). The programmer can ignore it, because the pre-compiler takes the responsibility to update the internal structures. The behavior for the E-methods can be defined by the programmer as he/she does for usual methods.

The usage of extenders is quite easy. As it can be noticed in Fig. 7, extenders are instantiated as usual classes, but for the special meaning of the first parameter in the constructor. Further, the extension-objects must be destroyed before the support-object. Note that this automatically happens if the storage class of the instance is **auto** (i.e. the allocation is performed on the stack) because the destruction order for the objects declared in a block is the reverse of the declaration order, according to the standard C++ semantic [19, 13].

ECO C++ supports the classer concept by means of the keyword **dynamic**. Fig. 8 shows a declaration for a classer. All constructors have to be private because the presence of the classer states that a given property holds. Here, we suggest to introduce a class method (declared **static**) that performs the test and eventually instantiates the classer, if admissible. We call such class method a pseudo-constructor. The programmer does not need to maintain references to classer instances. A special syntax makes easy to access methods of a classer and to test if a classer is instantiated for a given support-object:

[6]At least two design choices are possible to deal with operation that disconnect the graph: an exeption can be thrown or the graph can be dynamically reclassified as NotConnected.

```
class Graph: extensible
    {
    public:
        // a (non E-) method
        Vertex Add_New_Vertex();
            {
            call_e_method( Check_Add_New_Vertex() );
            call_e_method( Pre_Add_New_Vertex() );
            ...
            call_e_method( Post_Add_New_Vertex(v) );
            return v;
            }
        // some E-methods
        extend void Check_Add_New_Vertex()
        extend void Pre_Add_New_Vertex()
        extend void Post_Add_New_Vertex(Vertex v)
        ...
    };
```

```
class Labeling: extend Graph
    {
    public:
        // 'g' compulsory parameter
        Labeling(Graph& g) {};
        ~Labeling() {}

        // E-methods definitions
        void Post_Add_New_Vertex(Vertex v)
            { /*initialize with an empty label*/ }
        ...
            // other methods
        Set(Vertex, char*){}
        Get(Vertex);
        ...
    };
```

Figure 6: An example of declaration of a support-class Graph with an extender Labeling, written in ECO C++.

```
main()
    {
    Graph G;      // A graph
    ...
                          // Two labelings
    Labeling L1(G);                    // automatic allocation
    Labeling* L2= new Labeling(G);     // dynamic allocation
    ... // use 'L1' and 'L2' accessing their methods as usual
    delete L2;     // A dynamically allocated object has to be deleted, as usual.
    ...
    } // here L1 is deleted before G
```

Figure 7: An example of usage of the extender and the support-class declared in Fig. 7.

$support\_object.\{classer\}.method(\ldots)$	call the method
$support\_object.\{classer\}$	return true if the classer is instantiated.

Fig. 9 shows the usage of the classer declared in Fig. 8. It shows how to invoke a pseudo-constructor, how to test if a classer is instantiated, and how to preform a classer method calling.

Similar concepts are implemented in other systems, but they differ from ECO in substantial aspects. The concepts of *signal* and *slot*, in Qt [16], are a support for dynamic updating comparable to the E-methods system, but they were born in a quite different field (GUI object component) where no needs for complex classification arise. So, signals and slots alone are not useful in solving the exposed classification problems. On the other hand, the Java member classes [8], are oriented to classification problems, but they lack of dynamic updating capabilities. In some sense, ECO represents a fusion of these two attracting approaches that permits completely new developments.

5.2 The run-time support of ECO C++

The run-time support for the ECO C++ language has to maintain low level structures that represent the relationships between each support-object and its current extension-objects. The following operations have to be performed using such structures:

- creation of an extension-object,
- deletion of an extension-object,

```
class Graph: extensible
    {
    . . .
    extend void Check_Add_Edge(); // An E-method.
    };

class PlanarGraph: dynamic Graph
    {
    private:
        PlanarGraph( Graph& ) {. . .};  // Client cannot call this constructor.

    public:
        static void Test(Graph& g);       // The pseudo-constructor performs the instantiation (if admissible).
        void Check_Add_Edge() {. . .};    // Check if planarity is preserved and throw and exception if needed
        int Get_number_of_faces();
    };
```

Figure 8: A delcaration of a classer for managing the planarity property.

```
main()
     {
     Graph g;
     . . .
     PlanarGraph::Test(g);  // planarity test (using pseudo-consructor)

     if ( g.{PlanarGraph} )      // is 'g' recognized as planar?
            {
            cout << "g is planar, # of faces="
                   << g.{PlanarGraph}.Get_number_of_faces()  // classer method calling
                   << endl;
            }
     } // all classers of 'g' are automatically destroyed
```

Figure 9: An example of usage of the classer declared in Fig. 8.

- invocation of an E-method.

It is easy to implement a support system that is based on linked lists and performs a creation and a deletion taking $O(1)$ time.

On the contrary, the invocation of an E-method takes $O(k)$ time, where k is the number of the current extension-objects, if the E-method execution takes $O(1)$ for each extension-object, which is the most common situation. However, k is most of the time only dependent on the algorithm, as it arises from the analysis that we made in Section 4.2.

6 Conclusions

Developing libraries of algorithms is a complex task requiring expertise both in algorithmics and in software engineering.

In this paper we have presented techniques that are mainly related to solving graph classification problems in libraries of graph algorithms. The introduced representation primitives allow the designer to model dynamically changing properties of graphs with little effort. They permit to overcome most of the limitations that come out with standard object-orientation, and to reach many of its aimed goals, like extensibility (the open-closed principle) and flexibility (reuse of preexistent modules in different contexts). Further, the primitives allow to elegantly (in our opinion) express known concepts like the *observer pattern* [7] and the *data accessor* [18].

The pre-compiler for the ECO C++ language permits to easily apply the new concepts avoiding many of the drawbacks that typically arise when using new paradigms. The pre-compiler is available on the Web at the address below:

`http://www.dia.uniroma3.it/˜pizzonia/eco`

The presented ideas have born in the Graph Drawing area and specifically during the development and usage of the GDToolkit [20] system and have already been applied within the same project.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[2] D. Alberts, G. Cattaneo, G. F.Italiano, U. Nanni, and C. D. Zaroliagis. A software library of dynamic graph algorithms. In R. Battiti and A. A. Bertossi, editors, *Proceedings of "Algorithms and Experiments" (ALEX98) Trento, Italy, February 9-11, 1998*, pages 129–136, 1998.

[3] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: an annotated bibliography, June 1994.

[4] Jonathan Berry, Nathaniel Dean, Mark Goldberg, Gregory Shannon, and Steven Skiena. Graph drawing and manipulation with LINK. In G. Di Battista, editor, *Graph Drawing (Proc. GD' 97)*, volume 1353 of *Lecture Notes Comput. Sci.*, pages 425–437. Springer-Verlag, 1998.

[5] Paola Bertolazzi, Giuseppe Di Battista, and Giuseppe Liotta. Parametric graph drawing. *IEEE Transactions on Software Engineering*, 21(8):662–673, August 1995.

[6] C. Friedrich. The ffGraph library. Manuscript, Lehrstuhl für Informatik, Univ. of Passau, December 1995.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, Reading, 1996.

[8] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[9] Barbara Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1988. Revised version of the keynote address given at OOPSLA '87.

[10] K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *Commun. ACM*, 38(1):96–102, 1995.

[11] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., 1988.

[12] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, 1991.

[13] Bjarne Stroustrup. *The C++ Programming Language: Second Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1991.

[14] R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1):61–79, January/February 1988.

[15] Roberto Tamassia and Luca Vismara. A case study in algorithm engineering for geometric computing. Technical Report CS-97-18, Department of Computer Science, Brown University, December 1997. Wed, 24 Jun 1998 13:22:55 GMT.

[16] Troll Tech. Qt: a GUI Software Toolkit, 1998. `http://www.troll.no/`.

[17] Peter Wegner. The object-oriented classification paradigm. In Peter Wegner and Bruce Shriver, editors, *Research Directions in Object-Oriented Programming*. The MIT Press, 1987.

[18] Karsten Weihe. Reuse of algorithms: Still a challenge to object-oriented programming. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*, volume 32, 10 of *ACM SIGPLAN Notices*, pages 34–48, New York, October5–9 1997. ACM Press.

[19] ANSI X3J16. American national standard for information systems — programming language — C++. Approved standard, ANSI.

[20] ALCOM-IT – GDToolkit, 1998. Terza Università di Roma.
`http://www.inf.uniroma3.it/people/gdb/wp12/`.